第2章 IA-32处理器基本功能

2.1 IA-32处理器简介
2.2 通用寄存器
2.3 标志寄存器
2.4 段寄存器
2.5 寻址方式

2.6 指令指针寄存器和简单控制转移

2.7 堆栈和堆栈操作

2.6 指令指针寄存器和简单控制转移

2.6.1 指令指针寄存器

2.6.2 常用条件转移指令

2.6.3 比较指令和数值大小比较

2.6.4 简单无条件转移指令

2.6.1 指令指针寄存器

➢指令指针寄存器

- ✓ IA-32系列CPU有一个32位的指令指针寄存器EIP
- ✓ 它是早先8086CPU指令指针寄存器IP的扩展
- ✓ 由CS和EIP确定所取指令的存储单元地址。CS给出当前代码段的段号，指令指针寄存器EIP给出偏移
- ✓ 如果代码段起始地址是0，则EIP给出的偏移，或者说有效地址，直接决定所取指令的存储单元地址
- ✓ 实方式下，段的最大范围是64K，EIP中的高16位必须是0，相当于只有低16位的IP起作用

2.6.1 指令指针寄存器

➢顺序执行指令的过程

✓CPU执行代码（程序）就是一条接一条地执行机器指令。可以把CPU执行指令的过程看作一条处理指令的流水线，其第一步是从存储器中取出指令。

✓在取出一条指令后，会根据所取指令的长度，**自动调整指令指针寄存器EIP的值，使其指向下一条指令**。这样，就实现了**顺序执行指令**。

2.6.1 指令指针寄存器

➢控制转移指令

✓所谓**转移**指，非自动顺序调整EIP内容

✓**控制转移指令**就是专门用于改变**EIP**内容的指令

✓多种控制转移指令：

- 条件转移指令
- 无条件转移指令
- 循环指令
- 函数调用及返回指令等

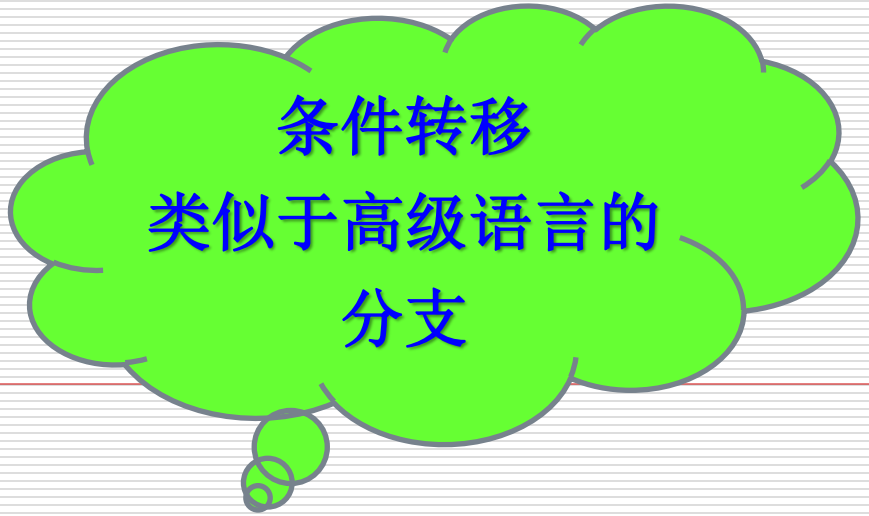
✓各种控制转移指令能够用于根据不同的情形改变**EIP**内容，从而实现转移

2.6.2 常用条件转移指令

➢条件转移

✓所谓**条件转移**指，当某一条件满足时，发生转移，否则继续顺序执行。换句话说，当某一条件满足时，就改变**EIP**的内容，从而实施转移，否则顺序执行。

✓**标志寄存器中的状态标志被用于表示条件**。绝大部分条件转移指令根据某个标志或者某几个标志来判断条件是否满足



2.6.2 常用条件转移指令

➢条件转移

✓根据一个标志判别

✓根据两个标志判别

✓根据三个标志判别

```
JC      LAB1          ; Jump if carry          ( CF=1 )
JBE     LAB2          ; Jump if below or equal ( CF=1 或 ZF=1 )
JLE     LAB3          ; Jump if less or equal  ( ZF=1 或 SF≠OF )
```

2.6.2 常用条件转移指令

➢条件转移指令（Jcc）

✓Jcc指令的一般格式

```
Jcc    LAB
```

符号cc表示各种条件缩写，LAB代表源程序中的标号。

当条件满足时，转移到标号LAB处；否则继续顺序执行。

条件转移指令是使用得最多的控制转移指令。

2.6.2 常用条件转移指令

➢条件转移指令（Jcc）

✓注意：同一指令，可能有多个助记符

```
JB      LABEL3          ;Jump if below
JNAE    LABEL3          ;Jump if not above or equal
JC      LABEL3          ;Jump if carry
```

2.6.2 常用条件转移指令

➢条件转移指令（Jcc）

指令格式	转移条件	转移说明	其他说明
JZ 标号 JE 标号	ZF=1 同上	等于0转移（Jump if zero） 相等转移（Jump if equal）	单个标志
JNZ 标号 JNE 标号	ZF=0 同上	不等于0转移（Jump if not zero） 不相等转移（Jump if not equal）	单个标志
JB 标号 JNAE 标号 JC 标号	CF=1 同上 同上	低于转移 不高于等于转移 进位位被置转移	单个标志 （无符号数）
JNBE 标号 JA 标号	(CF或ZF)=0 同上	不低于等于转移 高于转移	两个标志 （无符号数）
JLE 标号 JNG 标号	((SF异或OF)或ZF)=1 同上	小于等于转移 不大于转移	三个标志 （有符号数）
JNLE 标号 JG 标号	((SF异或OF)或ZF)=1 同上	不小于等于转移 大于转移	三个标志 （有符号数）

2.6.2 常用条件转移指令

➢条件转移指令（Jcc）

把寄存器ECX中的值视为有符号数。如下指令片段的功能：
当ECX中的值为0时，使EAX为0；
当ECX为正数时，使EAX为1；否则使EAX为-1。

```
    SUB    ECX, 0          ;在不改变ECX值的同时，根据ECX的值影响标志
    MOV    EAX, 0          ;先假设ECX值为0
    JZ     OVER            ;如果ZF=1（表示确实为0），转移到标号OVER处
    MOV    EAX, 1          ;再假设ECX值为正
    JNS    OVER            ;如果SF=0（表示确实为正），转移
    MOV    EAX, -1         ;至此，ECX值为负
OVER:
```

2.6.2 常用条件转移指令

➢说明

✓条件转移指令本身不影响标志。

✓条件转移指令在条件满足的情况下，只改变指令指针寄存器EIP。也就是说，条件转移的转移目的地仅限于同一个代码段内。这种不改变代码段寄存器CS，仅改变EIP的转移被称为**段内转移**。

✓条件转移指令可以实现向前方转移，也可以实现向后方转移。

今后将进一步介绍条件转移指令

2.6.2 常用条件转移指令

➢演示程序dp213

```
#include  <stdio.h>
int  arri[] = {23, 56, 78, 82, 77, 35, 22, 18, 44, 67};
int  main( )
{
    int  sum;  //用于存放累加和

    //嵌入汇编
    _asm  {
     .. .. ..
     .. .. ..
    }

    printf("sum=%d\n", sum);        //显示为sum=502
    return  0;
}
```

计算整型数组**arri**中
10个元素值之和

2.6.2 常用条件转移指令

➢演示程序dp213

计算整型数组arri中
10个元素值之和

```
_asm  {
      MOV     EAX, 0              //用于存放累加和
      MOV     ESI, 0              //作为数组的下标（索引）
      MOV     ECX, 10             //作为计数器
      LEA     EBX, arri           //得到数组首元素的有效地址
   NEXT:
      ADD     EAX, [EBX+ESI*4]    //累加某个元素值（由索引确定）
      INC     ESI                 //调整下标
      DEC     ECX                 //计数器减1（该指令会影响状态标志）
      JNZ     NEXT                //当ECX不为0，则从NEXT处继续执行
      ;
      MOV     sum, EAX            //保存累加和
}
```

2.6.3 比较指令和数值大小比较

➢比较指令（CMP）

✓比较指令的一般格式

CMP　DEST，SRC

根据**DEST – SRC**的差影响标志寄存器中的各状态标志，但不把作为结果的差送的目的操作数**DEST**。

除了不把结果送到目的操作数外，这条指令与**SUB**指令一样。
两个操作数的尺寸必须一致。

2.6.3 比较指令和数值大小比较

➢比较指令（CMP）

✓使用举例

```
CMP    EDX, -2                ;把EDX与-2比较
CMP    ESI, EBX               ;把ESI与EBX比较
CMP    AL, [ESI]              ;AL与由ESI所指的字节存储单元值作比较
CMP    [EBX+EDI*4+5], DX      ;由EBX+EDI*4+5所指字存储单元值与DX作比较
```

2.6.3 比较指令和数值大小比较

➢比较数值大小

✓为了比较两个数值的大小，一般使用比较指令。

根据零标志**ZF**是否置位，判断两者是否相等；

如果两者是无符号数，可根据进位标志**CF**判断大小；

如果两者是有符号数，要同时根据符号标志**SF**和溢出标志**OF**判断大小。

✓为了方便进行数值大小比较，**IA-32**系列**CPU**提供两套以数值大小为条件的条件转移指令，一套适用于无符号数之间的比较，另一套适用于有符号数之间的比较。这两套条件转移指令判断的标志是不同的。

有符号数间的次序关系称为大于**(G)**、等于**(E)**和小于**(L)**；

无符号数间的次序关系称为高于**(A)**、等于**(E)**和低于**(B)**。

在使用时要注意区分它们，不能混淆。

2.6.3 比较指令和数值大小比较

➢比较数值大小

设ECX和EDX含有两个数，现要求把较大者保存在ECX中，较小者保存在EDX中。

如果这两个数是有符号数，则代码片段如下：

```
    CMP    ECX, EDX
    JGE    OK                    ;有符号数比较大小转移（判断SF和OF）
    XCHG   ECX, EDX
OK:
```

如果这两个数是无符号数，则代码片段如下：

```
    CMP    ECX, EDX
    JAE    OK                    ;无符号数比较大小转移（判断CF）
    XCHG   ECX, EDX
OK:
```

2.6.3 比较指令和数值大小比较

➢比较数值大小

查看如下函数cf214的目标代码：

```
int   _fastcall   cf214(int x, int y)
{
   int  z = 1;
   if  ( x >= 13  &&  y <= 28 )
      z = 2;
   return  z;
}
```

调用约定 **_fastcall**

寄存器传递参数

2.6.3 比较指令和数值大小比较

➢比较数值大小

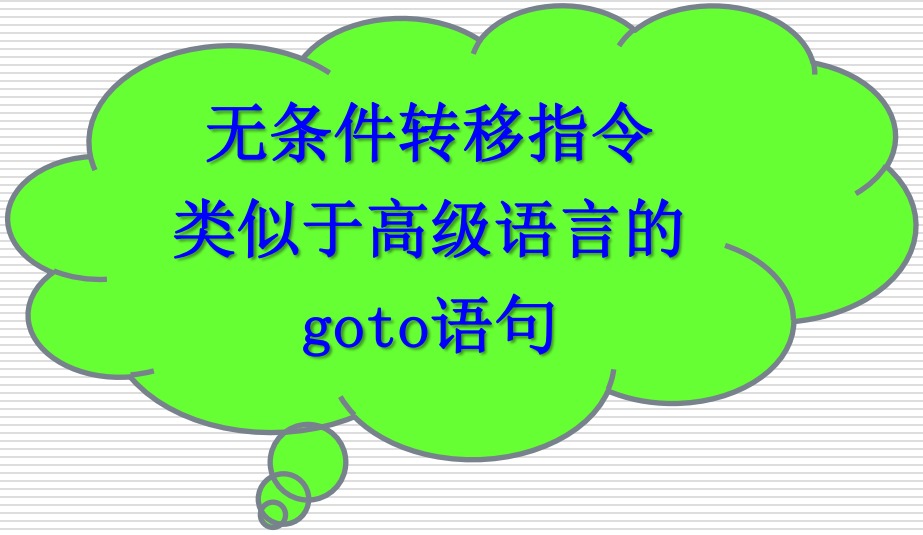
函数cf214的目标代码：

ECX传递x ， EDX传递y
EAX作为变量z

```
    mov    eax, 1
    cmp    ecx, 13                ;x与13比较
    jl     SHORT  LN1cf214        ;小于，则转移
    cmp    edx, 28                ;y与28比较
    jg     SHORT  LN1cf214        ;大于，则转移
    mov    eax, 2
LN1cf214:
    ret                           ;返回到调用者，函数cf23结束返回
```

符号“SHORT”表示转移目的地就在附近

2.6.4 简单无条件转移指令



2.6.4 简单无条件转移指令

➢无条件转移指令（JMP）

✓ 无条件转移指令的一般格式

JMP　LAB

指令使控制无条件地转移到标号**LAB**位置处。

所谓无条件是指没有任何前提，肯定实施转移。

无条件　段内　直接　转移指令

2.6.4 简单无条件转移指令

➢无条件转移指令（直接、段内）

把寄存器ECX中的值视为无符号数。当ECX中的值大于等于3时，使EAX为5；否则使EAX为7。

```
    CMP    ECX, 3          ；比较ECX和3
    JAE    LAB1            ；如ECX>=3，转移到标号NEXT处
    MOV    EAX, 7          ；否则，ECX=7
    JMP    LAB2            ；无条件转移到LAB2处
LAB1:
    MOV    EAX, 5
LAB2:
```

效率上，这样的代码片段并不好；
可读性，可以接受

2.6.4 简单无条件转移指令

➢示例

查看如下函数cf215的目标代码：

```
int  _fastcall  cf215(int x, int y)
{
   int  z;
   if  ( x > 10 )          //语句A
      z = 3*x+4*y+7;
   else
      z = 2*x+7*y-12;
   if ( y <= 20 )          //语句B
      z = 4*z+3;
   return  z;              //语句C
}
```

2.6.4 简单无条件转移指令

ECX传递x ， EDX传递y
EAX作为变量z

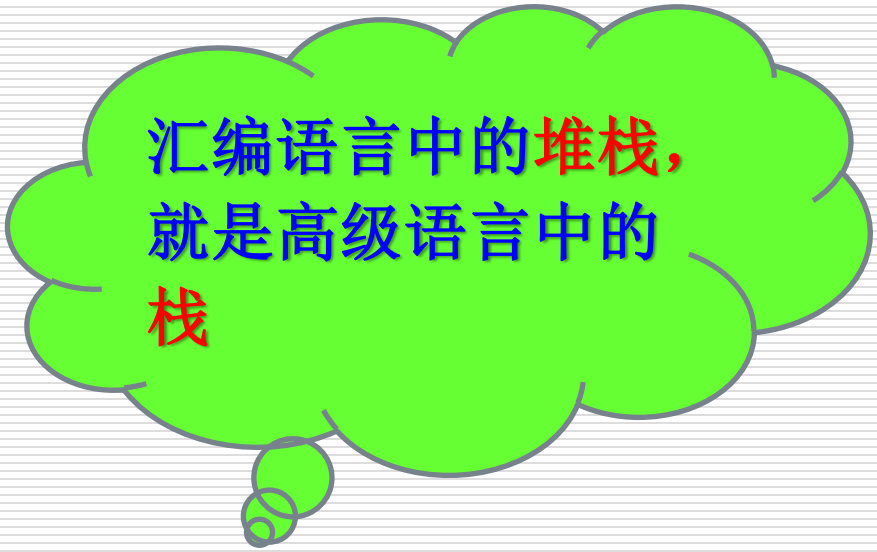
➢示例

```
;函数cf215目标代码（使速度最大化）
    cmp   ecx, 10                          ;x与10比较
    jle   SHORT  LN3cf215                  ;当小于等于10时转移
    lea   eax, DWORD PTR [ecx+ecx*2]       ;计算表达式3*x+4*y+7
    lea   eax, DWORD PTR [eax+edx*4+7]
    jmp   SHORT  LN2cf215                  ;无条件转（if-else语句结束）
LN3cf215:
    lea   eax, DWORD PTR [edx*8]           ;计算表达式7*y+2*x-12
    sub   eax, edx
    lea   eax, DWORD PTR [eax+ecx*2-12]
LN2cf215:
    cmp   edx, 20                          ;y与20比较
    jg    SHORT  LN1cf215                  ;当大于20时转
    lea   eax, DWORD PTR [eax*4+3]         ;计算4*z+3LN1cf215:
LN1cf215:
    ret                                    ;函数cf24结束返回
```

2.7 堆栈和堆栈操作

2.7.1 堆栈

2.7.2 堆栈操作指令



2.7.1 堆栈

➢堆栈

- ✓ 程序的运行与堆栈有密切关系：
  - **CPU**在运行程序期间往往需要利用堆栈保存某些关键信息
  - 程序自身也经常会利用堆栈临时保存一些数据
- ✓ 所谓**堆栈**其实就是一段内存区域，只是对它的访问操作仅限于一端进行。地址较大的一端被称为**栈底**，地址较小的一端被称为**栈顶**。

2.7.1 堆栈

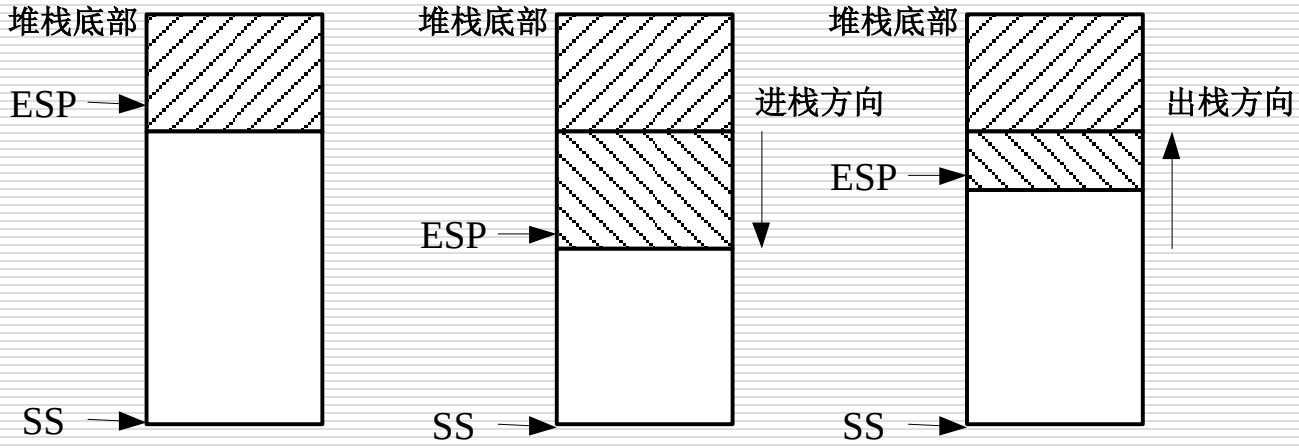
➢堆栈

- ✓ 堆栈操作遵守“后进先出”的原则，所有数据的存入和取出都在栈顶进行。
- ✓ 把存入数据的操作称为**进栈操作**，把取出数据的操作称为**出栈操作**。进栈操作也称为**压栈操作**，出栈操作也称为**弹出操作**。

2.7.1 堆栈

➢堆栈

- ✓ 堆栈段寄存器**SS**含有当前堆栈段的段号，**SS**指示堆栈所在内存区域的位置
- ✓ **堆栈指针寄存器ESP含有栈顶的偏移**（有效地址），**ESP指向栈顶**



(a)堆栈初始情形　(b)进栈后的情形　(c)部分出栈后的情形

2.7.1 堆栈

➢堆栈的用途

✓ 堆栈有如下所列的主要用途：

（1）保护寄存器内容或者保护现场；

（2）保存返回地址；

（3）传递参数；

（4）安排局部变量或者临时变量。

2.7.2 堆栈操作指令

- 进栈指令**PUSH**
- 出栈指令**POP**
- 通用寄存器全进栈指令和全出栈指令

2.7.2 堆栈操作指令

➢进栈指令PUSH

✓进栈指令的一般格式

```
PUSH   SRC
```

指令把源操作数**SRC**压入堆栈。

源操作数**SRC**可以是**32**位通用寄存器、**16**位通用寄存器和段寄存器，也可以是双字存储单元或者字存储单元，还可以是立即数。

把一个双字数据压入堆栈时，先把**ESP**减**4**，然后再把双字数据送到**ESP**所指示的存储单元。

把一个字数据压入堆栈时，先把**ESP**减**2**，再把字数据送到**ESP**所指示的存储单元。

ESP总是指向栈顶。

2.7.2 堆栈操作指令

➢进栈指令PUSH

✓使用举例

```
PUSH    EAX                 ;把EAX的内容压入堆栈
PUSH    DWORD PTR [ECX]     ;把ECX指示的双字存储单元的内容压入堆栈
PUSH    BX                  ;把BX的内容压入堆栈
PUSH    WORD PTR [EDX]      ;把EDX指示的字存储单元的内容压入堆栈
```

符号“**DWORD PTR**”表示双字存储单元

符号“**WORD PTR**” 表示字存储单元

至少进栈一个字！

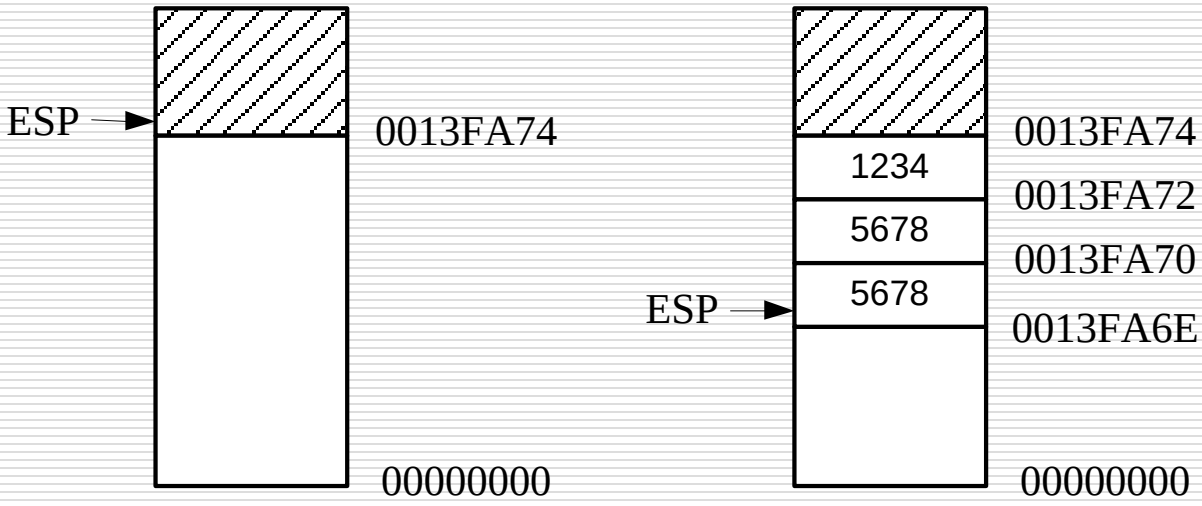
2.7.2 堆栈操作指令

➢进栈指令PUSH

✓使用举例

```
MOV     EAX, 12345678H
PUSH    EAX                 ;ESP=0013FA70H
PUSH    AX                  ;ESP=0013FA6EH
```

为节省篇幅，假设每一格为一个字存储单元（**16**位）



(a)堆栈初始情形　　**(b)**进栈后的情形

2.7.2 堆栈操作指令

➢出栈指令POP

✓出栈指令的一般格式

```
POP   DEST
```

指令从栈顶弹出一个双字或者字数据到目的操作数DEST。

目的操作数可以是32位通用寄存器、16位通用寄存器和段寄存器，也可以是字存储单元或者双字存储单元。如果目的操作数是双字的，那么就从栈顶弹出一个双字数据；否则，从栈顶弹出一个字数据。

注意：
出栈指令操作数不可以是立即数，
也不能是代码段寄存器CS。

2.7.2 堆栈操作指令

➢出栈指令POP

✓出栈指令的一般格式

```
POP   DEST
```

从栈顶弹出一个双字数据时，先从**ESP**所指示的存储单元中取出一个双字送到目的操作数，然后把**ESP**加**4**。

从栈顶弹出一个字数据时，先从**ESP**所指示的存储单元中取出一个字送到目的操作数，然后把**ESP**加**2**。

ESP总是指向栈顶。

2.7.2 堆栈操作指令

➢出栈指令POP

✓使用举例

```
POP    ESI                    ;从堆栈弹出一个双字到ESI
POP    DWORD PTR [EBX+4]      ;从堆栈弹出一个双字到EBX+4所指示存储单元
POP    DI                     ;从堆栈弹出一个字到DI
POP    WORD PTR [EDX+8]       ;从堆栈弹出一个字到EDX+8所指示的存储单元
```

符号“**DWORD PTR**”表示双字存储单元

符号“**WORD PTR**” 表示字存储单元

至少出栈一个字！

2.7.2 堆栈操作指令

➢示例

演示程序**dp216**及其嵌入汇编代码片段，
演示堆栈操作和堆栈指针寄存器变化。

```
#include  <stdio.h>
int  main( )  {
    int  varsp1, varsp2, varsp3, varsp4, varsp5;  //用于存放ESP值
    int  varr1, varr2;                            //用于存放EBX值
    _asm {    //嵌入汇编
        。。。。。。
    }
    printf("ESP1=%08XH\n",varsp1);     //显示为ESP1=0013FA74H
    printf("ESP2=%08XH\n",varsp2);     //显示为ESP2=0013FA70H
    printf("ESP3=%08XH\n",varsp3);     //显示为ESP3=0013FA6EH
    printf("ESP4=%08XH\n",varsp4);     //显示为ESP4=0013FA72H
    printf("ESP5=%08XH\n",varsp5);     //显示为ESP5=0013FA74H
    printf("EBX1=%08XH\n",varr1);      //显示为EBX1=56785678H
    printf("EBX2=%08XH\n",varr2);      //显示为EBX2=56781234H
    return  0;
}
```

2.7.2 堆栈操作指令

➢示例

```
_asm {
    MOV   EAX, 12345678H      //初值
    MOV   varsp1, ESP         //保存演示之初的ESP（假设为0013FA74H）
    ;
    PUSH  EAX                 //把EAX压入堆栈
    MOV  varsp2, ESP          //保存当前ESP（0013FA70H）
    ;
    PUSH  AX                  //把AX压入堆栈
    MOV   varsp3, ESP         //保存当前ESP（0013FA6EH）
    ;
    POP   EBX                 //从堆栈弹出双字到EBX
    MOV   varsp4, ESP         //保存当前ESP（0013FA72H）
    MOV   varr1, EBX
    ;
    POP   BX                  //从堆栈弹出字到BX
    MOV   varsp5, ESP         //保存当前ESP（0013FA74H）
    MOV   varr2, EBX
}
```

仅仅是示例！

2.7.2 堆栈操作指令

➢示例

演示堆栈用途之一，保护寄存器内容。

```
PUSH    EBP                ；保护EBP
PUSH    ESI                ；保护ESI
PUSH    EDI                ；保护EDI
.........                  ；其他操作
.........                  ；其间会破坏EBP、ESI和EDI的原有值
POP     EDI                ；恢复EDI
POP     ESI                ；恢复ESI
POP     EBP                ；恢复EBP
```

必须充分注意堆栈操作“后进先出”
同时确保堆栈平衡！

2.7.2 堆栈操作指令

- 16位通用寄存器全进栈、出栈指令
- 32位通用寄存器全进栈、出栈指令

有时需要把多个通用寄存器压入堆栈，以保护这些通用寄存器中的值。

为了提高效率，从**Intel 80186**开始，提供了通用寄存器全进栈指令和全出栈指令。

2.7.2 堆栈操作指令

➢16位通用寄存器全进栈、出栈指令

✓16位通用寄存器全进栈指令

PUSHA

✓16位通用寄存器全出栈指令

POPA

✓**PUSHA**指令将**8**个**16**位通用寄存器的内容压入堆栈，压入顺序：

AX、CX、DX、BX、SP、BP、SI、DI

✓**POPA**指令从堆栈弹出内容，以**PUSHA**相反的顺序送通用寄存器

2.7.2 堆栈操作指令

➢32位通用寄存器全进栈、出栈指令

✓32位通用寄存器全进栈指令

PUSHAD

✓32位通用寄存器全出栈指令

POPAD

✓**PUSHAD**指令将**8**个**32**位通用寄存器的内容压入堆栈，压入顺序：

EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI

✓**POPAD**指令从堆栈弹出内容，以**PUSHA**相反的顺序送通用寄存器

2.7.2 堆栈操作指令

➢示例

程序**dp217**及其嵌入汇编代码，演示**PUSHAD**指令的执行效果，还演示另一种访问堆栈区域存储单元的方法。

```
#include  <stdio.h>
int    buff[8];          //全局数组，存放从堆栈中取出的各寄存器之值
int  main( )
{
    _asm {     //嵌入汇编
          。。。。。。
    }
    //依次显示数组buff各元素之值，从中观察PUAHAD指令压栈的效果
    int  i;
    for  (i=0; i<8; i++)
        printf("buff[%d]=%u\n", i, buff[i]);
    return  0;
}
```

2.7.2 堆栈操作指令

➢示例

保护重要寄存器！

```
_asm  {//嵌入汇编
    PUSH   EBP              //先保存EBP！！
    ;
    MOV    EAX, 0           //给各通用寄存器赋一个特定的值
    MOV    EBX, 1
    MOV    ECX, 2
    MOV    EDX, 3
    ;                       //决不能随意改变ESP！！
    MOV    EBP, 5
    MOV    ESI, 6
    MOV    EDI, 7
    ;
    PUSHAD                  //把8个通用寄存器之值全部推到堆栈
    ;
```

2.7.2 堆栈操作指令

➢示例

```
    MOV   EBP, ESP          //使得EBP也指向堆栈顶
    LEA   EBX, buff         //把数组buff首元素的有效地址送到EBX
    MOV   ECX, 0            //设置计数器（下标）初值
NEXT:
    MOV   EAX, [EBP+ECX*4]      //依次从堆栈中取
    MOV   [EBX+ECX*4], EAX      //依次保存到数组buff
    INC   ECX                   //计数器加1
    CMP   ECX, 8                //是否满8
    JNZ   NEXT                  //没有满8个，继续处理下一个
    ;
    POPAD                       //恢复8个通用寄存器
    POP   EBP                   //恢复开始保存的EBP
}
```

缺省SS堆栈！

恢复被保护寄存器，同时确保堆栈平衡！

2.7.2 堆栈操作指令

➢示例

从堆栈到数组图示

```
    MOV    EBP, ESP
    LEA    EBX, buff
    MOV    ECX, 0
NEXT:
    MOV    EAX, [EBP+ECX*4]
    MOV    [EBX+ECX*4], EAX
    INC    ECX
    CMP    ECX, 8
    JNZ    NEXT
```